

Besturingssystemen theorie

Academiejaar 2018 – 2019



Inhoudsopgave

- Hoofdstuk 1: Inleiding Linux
- Hoofdstuk 2: Inleiding Besturingssystemen
- Hoofdstuk 3: Scheduling
- Hoofdstuk 4: Scripting in Linux
- **Hoofdstuk 5: Concurrency – parallele processen**
- Hoofdstuk 6: Processen in Linux
- Hoofdstuk 7: I/O van een besturingssysteem

HO
GENT

Hoofdstuk 3: Concurrency – parallele processen

1. Wat is concurrency?
2. Wederzijdse uitsluiting (mutual exclusion)
3. Het programmeren van wederzijdse uitsluiting
4. Het algoritme van Dekker
5. Het algoritme van Peterson
6. Semaforen
7. Monitoren
8. Synchronisatie
9. Deadlock
10. Threads

1. Wat is concurrency?

- **Multiprogrammering:** het beheer van meerdere processen in een systeem met 1 processor
- **Multiprocessing:** het beheer van meerdere processen in een systeem met meerdere processors
- **Gedistribueerde verwerking:** het beheer van meerdere processen die worden uitgevoerd op een aantal verspreide computersystemen.

Aan de basis van al deze zaken, en daarmee aan de basis van het ontwerp van besturingssystemen, ligt **Concurrency (gelijktijdig, samenlopen)**.

Concurrency hangt samen met tal van ontwerpkwesties zoals: de communicatie tussen processen, het delen van en het vechten om bronnen, de synchronisatie van meerdere procesactiviteiten en het toedelen van processortijd aan processen.



Concurrency (ofwel parallelle processen) is bij computerprocessen een belangrijke plaats gaan innemen. Doordat het mogelijk werd enorme rekencapaciteit in een kleine chip te stoppen, zijn multiprocessors gemeengoed geworden. Dergelijke systemen kunnen vele taken gelijktijdig uitvoeren. Dit verhoogt de productiviteit, maar schept ook problemen.

1. Wat is concurrency? (2)

In de systemen met I/O channels (I/O-processors) zijn verscheidene acties tegelijkertijd gaande. De CPU werkt aan één proces, terwijl de I/O channels aan andere werken. Het is duidelijk dat het gebruik van meerdere processors de verwerkingscapaciteit vergroot.

Stel dat er een programmeertaal bestaat waarin je onafhankelijke processen kan specificeren, en dat er meerdere processors beschikbaar zijn om aan een proces te werken.

```
PARBEGIN
    statement1;
    statement2;
    ...
    statementn;
PAREND
```

Zulke talen bestaan:
Ada, Modula-2, Concurrent C, Concurrent Pascal, ...



1. Wat is concurrency? (3)

Concurrency treedt op in 3 verschillende situaties:

- **Meerdere toepassingen.** Multiprogrammering werd uitgevonden om verwerkingstijd dynamisch te kunnen verdelen tussen een aantal actieve toepassingen.
- **Gestructureerde toepassing.** Als uitbreiding op de beginselen van modular ontwerpen en gestructureerd programmeren kunnen sommige toepassingen effectief worden geprogrammeerd als een verzameling gelijktijdige processen.
- **Structuur van het besturingssysteem.** Dezelfde voordelen van het structureren gelden voor de systeemprogrammeur en we hebben gezien dat besturingssystemen zelf vaak worden geïmplementeerd als een verzameling processen of threads.



2. Wederzijdse uitsluiting (mutual exclusion)

2.1 Concurrency met meerdere processors



Niet alleen processen, maar ook activiteiten binnen één proces kunnen gelijktijdig worden uitgevoerd.

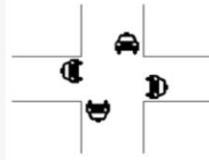
We zullen processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces. Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem.

De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

**HO
GENT**

Er bestaan verschillende niveaus van concurrency. Niet alleen processen, maar ook activiteiten binnen één proces kunnen gelijktijdig worden uitgevoerd. We zullen processen bespreken, maar de principes gelden eveneens voor activiteiten binnen één proces. Als parallelle processen niets gemeenschappelijk gebruiken, is er geen probleem. De moeilijkheden ontstaan wanneer de processen het gemeenschappelijke geheugen aanspreken.

2. Wederzijdse uitsluiting (mutual exclusion) (2)



2.2 Concurrency met 1 processor

→ Hier zijn er ook parallelle processen mogelijk.

In een dergelijk geval kunnen de processen niet tegelijkertijd worden uitgevoerd, maar ze kunnen wel op hetzelfde moment proberen de besturing van de CPU te krijgen.

Wanneer twee van zulke processen het gemeenschappelijk geheugen aanspreken, kunnen er nog steeds problemen ontstaan.

**HO
GENT**

2. Wederzijdse uitsluiting (mutual exclusion) (3)

Voorbeeld:

Beschouw een computersysteem met veel terminals. Stel dat de gebruikers elke regel, bestemd voor het computersysteem, beëindigen met de <enter>-toets. Stel dat wij het totaal aantal lijnen voor alle gebruikers samen wensen bij te houden in een variabele "LinesEntered". Veronderstel dat twee processen proberen de variabele "LinesEntered" simultaan te verhogen met 1.

Elk proces heeft dan zijn eigen kopij van volgende code:

```
load LinesEntered  
add 1  
store LinesEntered
```

Dit is een vervelende situatie: **de informatie in LinesEntered is fout!**



Dit is een vervelende situatie: de informatie in LinesEntered is fout! En bovendien wordt de fout veroorzaakt door een zwakke plek in het systeem dat toestaat dat twee processen op het verkeerde moment het gemeenschappelijke geheugen aanspreken. Merk op dat de twee processen het gemeenschappelijk geheugen niet op hetzelfde tijdstip benaderen. Door een wat ongelukkige timing ondermijnt het ene proces het andere. Het grootste probleem is dat bovenstaand voorbeeld 99,99% van de tijd misschien wel goed werkt. Het opsporen van dergelijke fouten kan daarom vrij moeilijk zijn.

2. Wederzijdse uitsluiting (mutual exclusion) (4)

2.3 Wederzijdse uitsluiting

→ de **kritieke sectie** van een proces is de code die naar gemeenschappelijke data verwijst

Als de uitvoering van een proces in de kritieke sectie is aangeland, moeten wij ervoor zorgen dat elk ander proces zijn eigen kritieke sectie niet betreedt. Omgekeerd moeten wij ook opletten dat een proces zijn kritieke sectie niet binnentreedt op het moment dat een ander proces in zijn kritieke sectie zit. Dit noemen we **wederzijdse uitsluiting (mutual exclusion)**.



We moeten processen gelijktijdig laten uitvoeren, en tegelijkertijd toch voorkomen dat bepaalde delen van die processen, de kritieke secties (critical sections), parallel worden verwerkt. Wanneer parallelle processen zich toegang verschaffen tot het gemeenschappelijke geheugen, bevatten hun kritieke secties de opdrachten die deze resources aanspreken. De kritieke sectie van een proces is dus de code die naar gemeenschappelijke data verwijst. Als de uitvoering van een proces in de kritieke sectie is aangeland, moeten wij ervoor zorgen dat elk ander proces zijn eigen kritieke sectie niet betreedt. Omgekeerd moeten wij ook opletten dat een proces zijn kritieke sectie niet binnentreedt op het moment dat een ander proces in zijn kritieke sectie zit. Dit noemen we wederzijdse uitsluiting (mutual exclusion).

Hoe kunnen we voor wederzijdse uitsluiting zorgen? Hoe garanderen we dat de uitsluiting er zal zijn, al vóór het proces zijn kritieke sectie heeft bereikt? Moeten we ook iets doen als een proces aan het einde van zijn kritieke sectie is?

2. Wederzijdse uitsluiting (mutual exclusion) (5)

2.3 Wederzijdse uitsluiting (vervolg)

Net voor de kritieke sectie van een proces wordt **ENTERMUTUALEXCLUSION** uitgevoerd en na de kritieke sectie wordt **EXITMUTUALEXCLUSION** uitgevoerd.

→ **ENTERMUTUALEXCLUSION** doet het volgende:

- controleren of een ander proces in zijn kritieke sectie is en, als dat het geval is, wachten;
- doorgaan met de uitvoering van de kritieke sectie als er geen ander proces in de kritieke sectie bezig is.

→ **EXITMUTUALEXCLUSION** moet alle andere processen vertellen dat een proces klaar is met de uitvoering van zijn kritieke sectie.



Net voor de kritieke sectie van een proces wordt **ENTERMUTUALEXCLUSION** uitgevoerd en na de kritieke sectie wordt **EXITMUTUALEXCLUSION** uitgevoerd. **ENTERMUTUALEXCLUSION** doet het volgende:

- controleren of een ander proces in zijn kritieke sectie is en, als dat het geval is, wachten;
- doorgaan met de uitvoering van de kritieke sectie als er geen ander proces in de kritieke sectie bezig is.

EXITMUTUALEXCLUSION moet alle andere processen vertellen dat een proces klaar is met de uitvoering van zijn kritieke sectie.

Met **ENTERMUTUALEXCLUSION** en **EXITMUTUALEXCLUSION** in een proces kunnen we van wederzijdse uitsluiting verzekerd zijn. Rest ons nog één probleem: hoe schrijven we dat? Hoe ziet dat eruit? Is het wel mogelijk zo'n code te creëren? Het schrijven van **ENTERMUTUALEXCLUSION** en **EXITMUTUALEXCLUSION** is niet zo eenvoudig.

Hoofdstuk 3: Concurrency – parallele processen

1. Wat is concurrency?
2. Wederzijdse uitsluiting (mutual exclusion)
3. Het programmeren van wederzijdse uitsluiting
4. Het algoritme van Dekker
5. Het algoritme van Peterson
6. Semaforen
7. Monitoren
8. Synchronisatie
9. Deadlock
10. Threads

3. Het programmeren van wederzijdse uitsluiting

We gaan uit van slechts twee gelijktijdige processen.

3.1 Eerste poging

We declareren een **boolaanse variabele “bezet”**, die voor beide processen globaal is. “Bezet” krijgt de waarde true als één van de processen zijn kritieke sectie ingaat en is false als dit niet het geval is.

Zo kan een proces dat aan zijn kritieke sectie moet beginnen, “bezet” controleren om te zien of het andere proces in zijn kritieke sectie is.

Het “wachten” en “wekken” kan op verschillende manieren worden geïmplementeerd. Een proces kan wachten en een ander proces kan het wekken.

Er is hier een probleem omdat de processen in ENTERMUTUALEXCLUSION gemeenschappelijk geheugen aanspreken.
Beide verwijzen naar “bezet”. Een ongelukkige timing kan tot gevolg hebben dat het ene proces het andere ondermijnt.



3. Het programmeren van wederzijdse uitsluiting (2)

3.2 Tweede poging

Eén manier om te voorkomen dat beide processen bijna gelijktijdig “bezet” controleren, is een tweede voorwaarde te gebruiken. We nemen aan dat beide processen op bijna hetzelfde moment proberen hun kritieke sectie in te gaan. Welk proces wanneer voorrang heeft, wordt geregeld door een globale variabele “welk”, met een waarde van of 1, of 2, te declareren. Beide processen moeten de waarde van “welk” controleren voordat zij hun kritieke secties ingaan. Eén proces mag doorgaan, het andere moet wachten. Zo wordt wederzijdse uitsluiting afgedwongen.

Helaas is er een ongewenst neveneffect. **De twee processen kunnen niet meer onafhankelijk worden uitgevoerd.** Ze moeten hun kritieke secties om beurten uitvoeren. Zo kan een proces door onbepaald uitstel worden getroffen: het moet voor onbepaalde tijd wachten. **Deze oplossing brengt wederzijdse uitsluiting tot stand, maar er moet wel veel voor worden ingeleverd.**
Processen worden misschien helemaal niet uitgevoerd.



3. Het programmeren van wederzijdse uitsluiting (3)

3.2 Derde poging

Zwakke plek in 2^{de} poging → gebruik van de variabele “welk” om te bepalen welk proces zijn kritieke sectie kan ingaan. De waarde van “welk” staat dit slechts aan één proces toe, zonder rekening te houden met wat het tweede proces aan het doen is. Dit is een te zware beperking.

Twee processen kunnen hun kritieke secties op hetzelfde moment ingaan omdat beide een “bezetting” pas claimen nadat gecontroleerd is of er een proces met zijn kritieke sectie bezig is.



Een zwakke plek in de tweede poging is het gebruik van de variabele “welk” om te bepalen welk proces zijn kritieke sectie kan ingaan. De waarde van “welk” staat dit slechts aan één proces toe, zonder rekening te houden met wat het tweede proces aan het doen is. Dit is een te zware beperking. We hadden iets nodig om ervoor te zorgen dat twee processen niet gelijktijdig hun kritieke secties ingaan, maar dit was duidelijk een verkeerde benadering.

Twee processen kunnen hun kritieke secties op hetzelfde moment ingaan omdat beide een “bezetting” pas claimen nadat gecontroleerd is of er een proces met zijn kritieke sectie bezig is. Anders gezegd: **een proces controleert eerst de waarde van de globale variabele “bezet”, en definieert deze dan pas als true.** Idee: verwissel deze twee opdrachten in ENTERMUTUALEXCLUSION.

3. Het programmeren van wederzijdse uitsluiting (4)

3.2 Derde poging (vervolg)

Zwakke plek → wanneer een proces “bezet” op true zet, moet het wachten omdat “bezet” true is. Het proces maakt het zichzelf onmogelijk in zijn kritieke sectie te komen. De poging mislukt omdat het hier **geen verschil maakt welk proces in zijn kritieke sectie zit**. Een proces moet onderscheid kunnen maken tussen zichzelf en andere processen.

Mogelijke oplossing → twee globale booleaanse variabelen gebruiken “bezet1” en “bezet2”. “Bezet1” is true als proces 1 in zijn kritieke sectie is en false als dit niet het geval is. “Bezet2” is true als proces 2 in zijn kritieke sectie is en false als het dat niet is. **Een proces declareert dus het betreden van zijn kritieke sectie en controleert dan of het andere proces dat ook heeft gedaan.**

Als proces 1 in zijn kritieke sectie is en deze vervolgens verlaat, zet het “bezet1” op false. Als proces 2 staat te wachten, wordt het hervat. Het omgekeerde vindt plaats als proces 2 zijn kritieke sectie verlaat. Elk proces kan zijn kritieke sectie dus diverse malen uitvoeren indien het ander proces inactief is.



3. Het programmeren van wederzijdse uitsluiting (5)

3.2 Derde poging (vervolg)

Deze constructie zorgt voor wederzijdse uitsluiting zonder de processen te dwingen bij toerbeurt hun kritieke secties in te gaan. Alleen als het andere proces niet in zijn kritieke sectie is of bezig is daar in te gaan, kan een proces zijn kritieke sectie betreden. **Wederzijdse uitsluiting is dus gegarandeerd.**

Helaas is er **nog een probleem** dat zich kan voordoen. Stel dat beide processen tegelijk, of vrijwel op hetzelfde moment hun kritieke sectie proberen in te gaan. Het probleem is dat elk proces denkt dat het andere in zijn kritieke sectie is. **Elk wacht dus tot de ander EXITMUTUALEXCLUSION heeft uitgevoerd. Omdat beide wachten, gebeurt er niets, nooit meer.**

Een dergelijke situatie, waarin twee processen elk erop wachten dat de ander iets doet, noemen we een **deadlock (impasse)**. Het resultaat is dat geen van de processen verder kan. Gewoonlijk moet één proces worden afgebroken, waarbij al het verrichte werk geheel of gedeeltelijk verloren gaat.

Dan moet het proces opnieuw worden gestart.



4. Het algoritme van Dekker

De Nederlandse wiskundige **Dekker** heeft een algoritme ontwikkeld dat wederzijdse uitsluiting **zonder ongewenste neveneffecten** garandeert.

Het staat in voor **wederzijdse uitsluiting voor twee parallelle, asynchrone processen** door ideeën uit de eerder beschreven algoritmen te gebruiken.



De drie pogingen tot wederzijdse uitsluiting illustreren de complexiteit van het probleem. Elk heeft een andere kwaal waar een systeem met parallelle processen kan aan lijden. Elk demonstreert ook een andere manier van denken om tot een oplossing te komen. De Nederlandse wiskundige Dekker heeft een algoritme ontwikkeld dat wederzijdse uitsluiting zonder ongewenste neveneffecten garandeert. Het staat in voor wederzijdse uitsluiting voor twee parallelle, asynchrone processen door ideeën uit de eerder beschreven algoritmen te gebruiken.

4. Het algoritme van Dekker (2)

```
int bezet1 = 0;      // false
int bezet2 = 0;      // false
int welk = 1;

int main(void)
{ // main
    void proces1 (void);
    void proces2 (void);
    parbegin
        proces1 ();
        proces2 ();
    parend
    return 0;
} // main
```



Het algoritme van Dekker komt in die zin met de derde poging overeen, dat twee booleaanse variabelen het binnengaan in een kritieke sectie aangeven. Het lijkt op de tweede poging waarin ook een globale variabele een ommezwaai in prioriteit aangeeft. Het algoritme lijkt ook op de eerste poging: elk proces controleert of een kritieke sectie is betreden voordat het probeert zijn eigen kritieke sectie in te gaan.

4. Het algoritme van Dekker (3)

```
void proces1 (void)
{
    // proces1
    ...
    // begin van ENTERMUTUALEXCLUSION
    bezet1 = 1;    // true
    while (bezet2)
        if  (welk == 2)
            {
                // if
                bezet1 = 0;  // false
                while (welk == 2);    // wacht tot welk 1 wordt
                bezet1 = 1;  // true
            }    // if
    // einde van ENTERMUTUALEXCLUSION
    ...
    // kritieke sectie van proces1
    ...
    // begin van EXITMUTUALEXCLUSION
    welk = 2;
    bezet1 = 0;    // false
    // einde van EXITMUTUALEXCLUSION
    ...
}
// proces1
```

4. Het algoritme van Dekker (4)

```
void proces2 (void)
{ // proces2
...
// begin van ENTERMUTUALEXCLUSION
bezet2 = 1; // true
while (bezet1)
    if (welk ==1)
        { // if
            bezet2 = 0; // false
            while (welk == 1); // wacht tot welk 2 wordt
            bezet2 = 1; // true
        } // if
// einde van ENTERMUTUALEXCLUSION
...
// kritieke sectie van proces2
...
// begin van EXITMUTUALEXCLUSION
welk = 1;
bezet2 = 0; // false
// einde van EXITMUTUALEXCLUSION
...
} // proces2
```



De logica achter het algoritme van Dekker is de volgende. Voordat een proces in zijn kritieke sectie gaat, moet het:

1. Zijn bezettingsvariabele op true zetten. Dit betekent dat het probeert zijn kritieke sectie in te gaan.
2. Controleren of het andere proces in zijn kritieke sectie is of probeert daarin te komen. Is dat niet het geval: kritieke sectie ingaan. Anders: verder gaan met de volgende stap.
3. Wachten als het andere proces aan de beurt is om zijn kritieke sectie uit te voeren. De bezettingsvariabele op false zetten en wachten tot het andere proces zijn kritieke sectie verlaat.
4. In het geval dat het huidige proces aan de beurt is om zijn kritieke sectie uit te voeren - als het andere proces toch in zijn kritieke sectie is: wachten tot het deze verlaat. Probeert daarentegen het andere proces ook zijn kritieke sectie in te gaan, dan moet het wachten, zodra het ontdekt dat het huidige proces aan de beurt is. In die situatie: de kritieke sectie ingaan.

4. Het algoritme van Dekker (5)

Er zijn twee grote verschillen tussen deze oplossing en de vorige.

1^{ste} → de booleaanse variabelen geven niet aan of een proces daadwerkelijk in zijn kritieke sectie is; ze maken slechts kenbaar dat een proces dat wil gaan doen.

2^{de} → de voorrang niet streng wordt voorgeschreven, tenzij beide processen op vrijwel hetzelfde moment proberen hun kritieke secties in te gaan.



4. Het algoritme van Dekker (6)

De logica achter het algoritme van Dekker →

Voordat een proces in zijn kritieke sectie gaat, moet het:

1. Zijn bezettingsvariabele op true zetten. Dit betekent dat het probeert zijn kritieke sectie in te gaan.
2. Controleren of het andere proces in zijn kritieke sectie is of probeert daarin te komen. Is dat niet het geval: kritieke sectie ingaan. Anders: verder gaan met de volgende stap.
3. Wachten als het andere proces aan de beurt is om zijn kritieke sectie uit te voeren. De bezettingsvariabele op false zetten en wachten tot het andere proces zijn kritieke sectie verlaat.
4. In het geval dat het huidige proces aan de beurt is om zijn kritieke sectie uit te voeren - als het andere proces toch in zijn kritieke sectie is: wachten tot het deze verlaat. Probeert daarentegen het andere proces ook zijn kritieke sectie in te gaan, dan moet het wachten, zodra het ontdekt dat het huidige proces aan de beurt is. In die situatie: de kritieke sectie ingaan.



5. Het algoritme van Peterson

→ biedt een eenvoudiger elegante oplossing

```
boolean flag [2];
int turn;
void P0()
{   while (true)
{   flag [0] = true;
    turn = 1;
    while (flag [1] && turn == 1)
        /* do nothing */;
    /* critical section */;
    flag [0] = false;
    /* remainder */
}
}
```

$A.p \wedge A.q$
≡ $x.p \wedge (\neg x.q \vee H.p.q) \wedge x.q \wedge (\neg x.p \vee H.q.p)$ {keuze voor A}
≡ $x.p \wedge H.p.q \wedge x.q \wedge H.q.p$ {predicaten calculus}
⇒ $H.p.q \wedge H.q.p$ {over de x-en weten we niets}
⇒ $false$ {dat willen we graag}

Het algoritme van Dekker lost het probleem van wederzijdse uitsluiting op, maar gebruikt daarvoor een nogal complex programma, dat moeilijk te volgen is en waarvan de juistheid lastig is te bewijzen.

De globale variabele **flag** duidt de positie van elk proces ten aanzien van wederzijdse uitsluiting aan.

De globale variabele **turn** lost de conflicten van gelijktijdigheid op,

5. Het algoritme van Peterson (2)

```
/* vervolg algoritme */
void P1()
{
    while (true)
    {   flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0)
            /* do nothing */;
        /* critical section */;
        flag [1] = false;
        /* remainder */
    }
}

void main()
{   flag [0] = false;
    flag [1] = false;
    parbegin (P0, P1);
}
```



5. Het algoritme van Peterson (3)

Bespreking algoritme:

De globale variabele **flag** duidt de positie van elk proces ten aanzien van wederzijdse uitsluiting aan.

De globale variabele **turn** lost de conflicten van gelijktijdigheid op.

De wederzijdse uitsluiting blijft hier behouden → wanneer P0 de flag[0] op true instelt, kan P1 zijn kritieke sectie niet uitvoeren. Bevindt P1 zich al in zijn kritieke sectie, dan geldt dat flag[1] = true en is de uitvoering van de kritieke sectie van P0 geblokkeerd.

**HO
GENT**

5. Het algoritme van Peterson (4)

Bespreking algoritme (vervolg):

Een wederzijdse blokkering wordt echter voorkomen.

Veronderstel dat P0 is geblokkeerd in zijn while lus. Dit betekent dat flag[1] true is en dat turn = 1. P0 kan zijn kritieke sectie uitvoeren als flag[1] false wordt of turn 0 wordt.

Beschouw nu 3 uitputtende gevallen:

1. **P1 heeft geen interesse in zijn kritieke sectie.** Dit geval is onmogelijk, omdat het impliceert dat flag[1] = false.
2. **P1 wacht op zijn kritiek sectie.** Dit geval is ook onmogelijk, omdat als turn = 1, P1 zijn kritieke sectie kan uitvoeren.
3. **P1 gebruikt zijn kritieke sectie herhaaldelijk en monopoliseert daarmee de toegang.** Dit kan niet gebeuren, omdat P1 aan P0 een kans moet geven door turn in te stellen op 0 voordat P1 probeert zijn kritieke sectie uit te voeren.

**HO
GENT**

5. Wederzijdse uitsluiting bij n processen

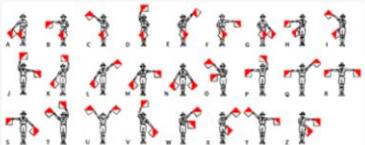
Het algoritme van Dekker is niet gemakkelijk op meer dan twee processen toe te passen. Daarvoor hebben we dus een ander algoritme nodig zoals bijvoorbeeld het algoritme van Peterson.

Er zijn er vele. Sommige garanderen dat een proces dat zijn kritieke sectie probeert uit te voeren, nooit erg lang wordt uitgesteld. Door hun complexiteit zijn deze algoritmen heel moeilijk in de praktijk toe te passen.

**HO
GENT**

Hoofdstuk 3: Concurrency – parallele processen

1. Wat is concurrency?
2. Wederzijdse uitsluiting (mutual exclusion)
3. Het programmeren van wederzijdse uitsluiting
4. Het algoritme van Dekker
5. Het algoritme van Peterson
6. Semaforen
7. Monitoren
8. Synchronisatie
9. Deadlock
10. Threads



6. Semaforen

6.1. Inleiding

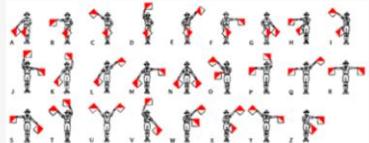
In 1965 introduceerde Dijkstra het begrip semafoor om wederzijdse uitsluiting tussen processen af te dwingen. Hij definiereerde een semafoor als **een integer-variabele die door slechts 2 primitieve operaties kan worden veranderd.**

Een primitieve kan niet worden onderbroken; eenmaal begonnen, kan het proces tot het klaar is niet worden onderbroken of opgeschort. Primitieve operaties zijn afhankelijk van het systeemontwerp, daarom moet er bij het ontwerpen van computersystemen al rekening mee worden gehouden.

**HO
GENT**

Ter info: De semafoor of optische telegraaf was het eerste, bruikbare middel voor optische telecommunicatie. Het woord semafoor is een samenstelling van de Griekse woorden σημα (teken) en φορεω (dragen)

Afbeelding: vlaggen semafoor → het alfabet



6. Semaforen (2)

6.1. Inleiding (vervolg)

De primitieve operaties voor een semafoor zijn P en V, en worden als volgt gedefinieerd. Als S een semafoor is, dan is

```
P(S): if (S > 0)
      S = S - 1;
      else (proces uitstellen)
```

```
V(S): if (een proces uitgesteld is als gevolg van P(S))
      (hervat een proces)
      else S = S + 1;
```

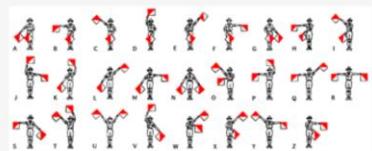
Een proces dat de primitieve P uitvoert, moet misschien wachten (WAIT). Een proces dat de primitieve V uitvoert, geeft misschien het signaal dat een ander proces kan worden hervat. Het feit dat P en V niet onderbroken kunnen worden is essentieel. Een eenmaal begonnen operatie eindigt zonder onderbreking.

**HO
GENT**

P → Prolaag "Prolaag" is een woord dat Dijkstra heeft bedacht en het betekent "probeer te verlagen".

V → Verhoog

6. Semaforen (3)



6.1. Inleiding (vervolg)

Het algoritme wordt dan:

```
int S = 1;           // semafoor
int main(void)
{
    //main
    void proces1 (void);
    void proces2 (void);
    parbegin
        proces1 ();
        proces2 ();
    parend
    return 0;
} //main
```

**HO
GENT**

6. Semaforen (4)

6.1. Inleiding (vervolg)

```
void proces1 (void)
{
    //proces1
    ...
    // begin van ENTERMUTUALEXCLUSION
    P(S);
    // einde van ENTERMUTUALEXCLUSION
    ...
    // kritieke sectie van proces1
    ...
    // begin van EXITMUTUALEXCLUSION
    V(S);
    // einde van EXITMUTUALEXCLUSION
    ...
}
```



6. Semaforen (5)

6.1. Inleiding (vervolg)

```
void proces2 (void)
{
    //proces2
    ...
    // begin van ENTERMUTUALEXCLUSION
    P(S);
    // einde van ENTERMUTUALEXCLUSION
    ...
    // kritieke sectie van proces2
    ...
    // begin van EXITMUTUALEXCLUSION
    V(S);
    // einde van EXITMUTUALEXCLUSION
    ...
}
```



6. Semaforen (6)

6.1. Inleiding (vervolg)

Naast de eenvoud en de elegantie van semaforen hebben ze nog een ander belangrijk voordeel. Het algoritme kan gemakkelijk worden uitgebreid voor een situatie geval met n parallelle processen. **Als 1 proces P(S) uitvoert, zijn alle andere gedwongen te wachten.**





6. Semaforen (7)

6.2. Sterke semaforen

Een **semafoon** is dus een onderdeel van een synchronisaties mechanisme voor parallelle of gedistribueerde programma's.

Het grondbeginsel luidt als volgt:

- Twee of meer processen kunnen samenwerken d.m.v. eenvoudige signalen, waarbij een proces kan worden gedwongen te stoppen op een opgegeven plaats totdat het een specifiek signaal heeft ontvangen.
- Voor het signaleren worden speciale variabelen gebruikt, zogenoemde **semaforen**.

Men kan aan elke coördinatie-eis, hoe complex ook, voldoen door de keuze van de juiste signaalstructuur.

**HO
GENT**

De eerste grote sprong voorwaarts bij het oplossen van problemen van gelijktijdige processen was het proefschrift van Dijkstra. Dijkstra hield zich bezig met het ontwerpen van een besturingssysteem als een verzameling samenwerkende, sequentiële processen en met het ontwikkelen van efficiënte en betrouwbare mechanisme voor het ondersteunen van de samenwerking. Deze mechanismen kunnen even gemakkelijk ook worden gebruikt door gebruiksprocessen als de processor en het besturingssysteem de mechanisme beschikbaar maken.

Het grondbeginsel luidt als volgt. Twee of meer processen kunnen samenwerken d.m.v. eenvoudige signalen, waarbij een proces kan worden gedwongen te stoppen op een opgegeven plaats totdat het een specifiek signaal heeft ontvangen. Voor het signaleren worden speciale variabelen gebruikt, zogenoemde semaforen. Men kan aan elke coördinatie-eis, hoe complex ook, voldoen door de keuze van de juiste signaalstructuur.

6. Semaforen (8)

6.2. Sterke semaforen (vervolg)

- voor het verzenden van een signaal via semafoor s voert een proces de primitieve **signal**(s) uit.
- voor het ontvangen van een signaal via semafoor s voert een proces de primitieve **wait**(s) uit; is het corresponderende signaal nog niet verzonden, dan wordt het proces onderbroken totdat het versturen ervan plaatsvindt.

Om het gewenste effect te bereiken kunnen we de semafoor beschouwen als een variabele die een gehele waarde heeft en waarvoor **3 bewerkingen zijn gedefinieerd**:

1. Een semafoor kan worden **geïnitialiseerd op een niet-negatieve waarde**.
2. De bewerking **wait verlaagt de semafoorwaarde**. Wordt de waarde negatief, dan wordt het proces dat de opdracht wait uitvoert, geblokkeerd.
3. De bewerking **signal verhoogt de semafoorwaarde**. Is de waarde niet positief, dan wordt een proces dat is geblokkeerd door een bewerking wait gedeblokkeerd.

Er bestaat geen mogelijkheid, anders dan deze 3 bewerkingen, om semaforen te inspecteren of te bewerken.



Voor het verzenden van een signaal via semafoor s voert een proces de primitieve signal(s) uit. Voor het ontvangen van een signaal via semafoor s voert een proces de primitieve wait(s) uit; is het corresponderende signaal nog niet verzonden, dan wordt het proces onderbroken totdat het versturen ervan plaatsvindt.

Om het gewenste effect te bereiken kunnen we de semafoor beschouwen als een variabele die een gehele waarde heeft en waarvoor 3 bewerkingen zijn gedefinieerd:

1. Een semafoor kan worden geïnitialiseerd op een niet-negatieve waarde.
2. De bewerking wait verlaagt de semafoorwaarde. Wordt de waarde negatief, dan wordt het proces dat de opdracht wait uitvoert, geblokkeerd.
3. De bewerking signal verhoogt de semafoorwaarde. Is de waarde niet positief, dan wordt een proces dat is geblokkeerd door een bewerking wait gedeblokkeerd.

Er bestaat geen mogelijkheid, anders dan deze 3 bewerkingen, om semaforen te inspecteren of te bewerken.

6. Semaforen (9)

Formele definitie van primitieven voor semaforen:

```
struct semaphore {  
    int count;  
    queueType queue;  
}  
  
void wait(semaphore s)  
{    s.count--;  
    if (s.count<0)  
    {        place this process in s.queue;  
        block this process  
    }  
  
void signal(semaphore s)  
{    s.count++;  
    if (s.count<=0)  
    {        remove a process P from s.queue;  
        place process P on ready list;  
    }  
}
```

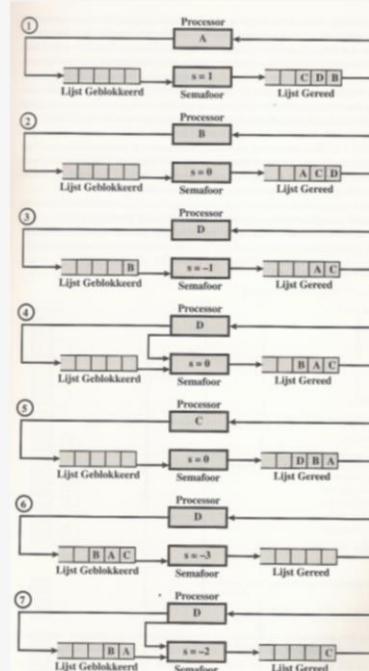


6. Semaforen (10)

Bevat de definitie van een semafoor deze FIFO-strategie, dan wordt dit een **sterke semafoon** genoemd.

Als niet is vastgelegd in welke volgorde processen uit de wachtrij worden verwijderd, is er sprake van een **zwakke semafoon**.

Voorbeeld van een sterke semafoon



Voorbeeld van een FIFO operatie:

Hierbij zijn de processen A,B en C afhankelijk van een resultaat van proces D.

In het begin (1) is A in uitvoering; B, C en D staan gereed; de semafoorteller is **gelijk aan 1**, wat aangeeft dat 1 resultaat van D beschikbaar is. Wanneer A een wait uitvoert, passeert het onmiddellijk de semafoon en kan doorgaan met de uitvoering; vervolgens kan het weer aansluiten in de rij gereed. **Vervolgens gaat B in uitvoering (2)**, voert ook een wait uit en wordt geblokkeerd; **dit geeft D de mogelijkheid voor uitvoering (3)**. Als D voor een nieuw resultaat heeft gezorgd, geeft het een signaal dat het mogelijk maakt dat **B verplaatst wordt naar de rij gereed (4)**. D sluit weer aan bij de rij gereed en **C komt in uitvoering (5)**, maar raakt geblokkeerd bij het geven van een wait. Op gelijke manier komen ook A en B in uitvoering en raken geblokkeerd op de semafoon, wat **D de gelegenheid geeft om verder te gaan met zijn uitvoering (6)**.

Wanneer D een resultaat heeft bereikt, geeft het een signaal zodat C verplaatst naar de rij gereed. Bij volgende optredens van D komen ook A en B uit de wachtrij van geblokkeerde processen.

6. Semaforen (11)

→ Het algoritme van wederzijdse uitsluiting:

```
/* program mutual exclusion */  
const int n = /* number of processes */;  
semaphore s = 1;  
void P(int i)  
{   while (true)  
    {      wait(s);  
          /* critical section */;  
          signal(s);  
          /* remainder */  
    }  
}  
Void main()  
{  
    parbegin (P(1), P(2), ... , P(n));  
}
```



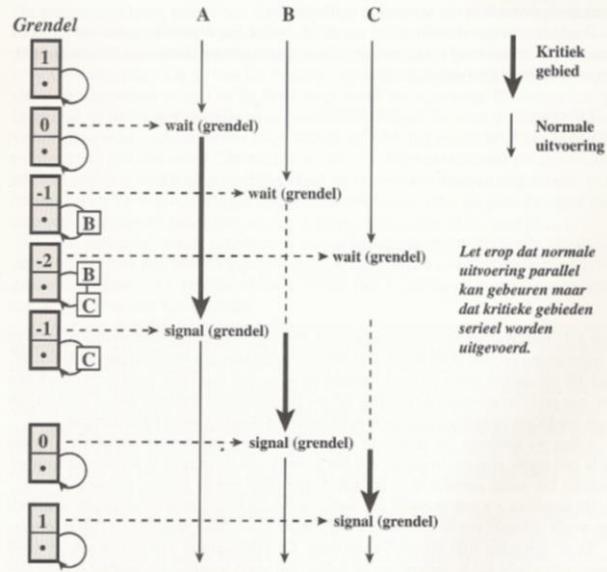
We zullen uitgaan van sterke semaforen omdat deze gemakkelijker zijn en omdat meestal besturingssystemen deze vorm van semaforen bieden,

Dit toont een eenvoudige oplossing van het probleem van wederzijdse uitsluiting met een semafoor s . Er zijn n processen, die worden aangegeven in de array $P(i)$. In elk proces wordt een $\text{wait}(s)$ uitgevoerd vlak voor de kritieke sectie. Wordt de waarde s negatief, dan wordt het proces opgeschort. Is de waarde 1, dan wordt deze verlaagd tot 0 en voert het proces onmiddellijk zijn kritieke sectie uit. Omdat s niet meer positief is, kan geen enkel ander proces zijn kritieke sectie uitvoeren.

De semafoor wordt geïnitialiseerd op 1. Daardoor kan het eerste proces dat een wait uitvoert, onmiddellijk zijn kritieke sectie binnengaan, waarbij de waarde van s gebracht wordt op 0. Elk proces dat probeert zijn kritieke sectie te betreden, merkt dat deze bezet is en wordt geblokkeerd, waarbij de waarde van s met 1 vermindert. Nog meer processen kunnen proberen toegang te krijgen: al deze pogingen leiden tot een verdere daling van de waarde van s . Verlaat het proces dat zijn kritieke sectie het eerst heeft gestart, de kritieke sectie, dan wordt s verhoogd en wordt een van de geblokkeerde processen (als er geblokkeerde processen zijn) verwijderd uit de wachtrij processen die geblokkeerd zijn voor de semafoor en in de toestand gereed geplaatst. Wordt het daarna ingeroosterd door het besturingssysteem, dan kan het zijn kritieke sectie uitvoeren.

6. Semaforen (12)

→ Mogelijke volgorde van 3 processen die voor de wederzijdse uitsluiting de aanpak van vorige slide gebruiken:



In dit voorbeeld benaderen 3 processen A,B en C een gedeelde bron die wordt bewaakt door de semafoor grendel. Proces A voert een wait(grendel) uit. Omdat de semafoorwaarde 1 is op het moment van de wait, kan A direct de kritieke sectie binnengaan en de semafoor krijgt de waarde 0. Terwijl A binnen zijn kritieke sectie zit, voeren B en C beiden een wait uit en worden geblokkeerd in afwachting van de beschikbaarheid van de semafoor. Wanneer A zijn kritieke sectie verlaat en signal(grendel) uitvoert, mag B zijn kritieke sectie binnengaan want B stond vooraan in de wachtrij.

6. Semaforen (13)

Implementatie semaforen:

1^{ste} mogelijkheid → implementeren in hardware of firmware

2^{de} mogelijkheid → softwarebenadering zoals algoritme van Dekker of Peterson => leidt tot een aanzienlijke overhead in de verwerking

3^{de} mogelijkheid → het gebruiken van een in hardware ondersteund mechanisme voor wederzijdse uitsluiting zoals bijvoorbeeld het gebruik van een instructie test and set waarbij de semafoor weer een datastructuur heeft en een nieuwe integer als component, s.flag bevat.
(zie volgende slide, als illustratie)

4^{de} mogelijkheid → bij een systeem met 1 processor is het mogelijk interrupts te verbieden tijdens de bewerkingen wait en signal.



Zoals eerder is gezegd, is het essentieel dat de bewerkingen wait en signal worden geïmplementeerd als atomaire primitieven. Een mogelijkheid die voor de hand ligt, is ze te implementeren in hardware of firmware. Is dat niet mogelijk, dan zijn er een aantal alternatieven. De kern van het probleem is de wederzijdse uitsluiting: slechts 1 proces tegelijk mag een semafoor wijzigen met de bewerking wait of signal. Daarvoor zou elk van de softwarebenaderingen kunnen worden gebruikt, bijvoorbeeld het algoritme van Dekker of het algoritme van Peterson; dit zou leiden tot een aanzienlijke overhead in de verwerking. Een ander alternatief is het gebruiken van een in hardware ondersteund mechanisme voor wederzijdse uitsluiting.

7. Monitoren

Semaforen zijn een primitief maar krachtig en flexibel gereedschap voor het afdwingen van wederzijdse uitsluiting en voor het coördineren van processen.

Het kan echter moeilijk zijn een correct programma te maken met semaforen. De moeilijkheid is dat de bewerkingen **wait** en **signal** verspreid kunnen zijn over het programma en het is lastig te zien welke algehele invloeden bewerkingen hebben op de betreffende semaforen.

De monitor is een constructie in een programmeertaal die een functionaliteit biedt die vergelijkbaar is met die van semaforen, maar die gemakkelijker te besturen is.

**HO
GENT**

7. Monitoren (2)

Om problemen te vermijden moet wederzijdse uitsluiting echter verplicht zijn.

Oplossing hiervoor is: de kritieke secties in een gebied te plaatsen waartoe maar 1 proces tegelijk toegang heeft. Dan gebruiken de processen de code op een manier die automatisch wederzijdse uitsluiting afdwingt. Voor de speciale gebieden gebruiken we de term **monitors**.

Een monitor is een constructie die code kan bevatten die naar gemeenschappelijke gebruikte data verwijst. Oppervlakkig gezien lijkt het een verzameling datatypen, datastructuren en procedures, onder de monitor heading. Maar een monitor is veel meer.

Hij bevat procedures en variabelen, maar de procedures zijn van een bijzonder type. Als parallelle processen verschillende procedures in een monitor aanroepen, dwingt de monitor de processen deze procedures na elkaar uit te voeren.



7. Monitoren (3)

2 procedures binnen dezelfde monitor kunnen niet tegelijk actief zijn. Door de taal gedefinieerde **aanroep-protocollen** (calling protocollen) dwingen dit automatisch af.

Een kritieke sectie kunnen we daarom, i.p.v. deze in een proces te coderen, als **monitor-procedure** schrijven. De code wordt dan niet geduplicateerd. Wanneer een proces gemeenschappelijke data moet gebruiken, roept het een monitor-procedure aan.

Door een compiler gegeneerde code, die de besturing aan een monitor-procedure overdraagt, wordt wederzijdse uitsluiting gegarandeerd.

Op deze manier is er een aanzienlijk verschil tussen een monitor en een eenvoudige collectie procedures. Een monitor dwingt heel streng wederzijdse uitsluiting af tussen processen die proberen zijn procedures uit te voeren. Om dit verschil te benadrukken wordt een monitor-procedure een **procedure-entry** genoemd.



7. Monitoren (4)

Via conditionele variabelen kan de monitor processen uitstellen of hervatten om events te synchroniseren.

Monitoren werken het best wanneer ze centraal geïnstalleerd kunnen worden. Omdat alle processen de monitor aanspreken, is deze het middelpunt van alle discussies en analyses. Maar vele systemen hebben geen centrale component.

De monitorconstructie is geïmplementeerd in enkele programmeertalen waaronder Modula-3, Java, Ook is ze geïmplementeerd als een programmabibliotheek. Dit biedt de mogelijkheid grenrels op elk object te plaatsen. Vooral bij zoiets als een verbonden lijst kan het zinvol zijn alle verbonden lijsten te vergrendelen met 1 grenrel, 1 grenrel te gebruiken voor elke lijst of 1 grenrel te gebruiken voor elk element van elke lijst.



7. Monitoren (5)

Een monitor kunnen we dus kort definiëren als een constructie in een programmeertaal die voorziet in abstracte gegevenstypen en toegang, met wederzijdse uitsluiting, tot een aantal procedures.



Wiki: Een monitor in het gedistribueerd programmeren is een synchronisatiemechanisme dat aangeboden wordt door de programmeertaal, het besturingssysteem of de hardware waarop/mee een multiprogramma uitgevoerd wordt.

De basis van alle synchronisatie is de semafoor, waarmee ieder synchronisatieprobleem fijnmazig op te lossen valt. Een monitor is een synchronisatie-mechanisme met een hoger abstractieniveau, waarin de synchronisatie op een minder fijnmazig niveau wordt uitgevoerd voor de programmeur in plaats van gepland door de programmeur zelf.

Om van een monitor gebruik te maken, moet een programmeur meestal een (deel van) zijn programma speciaal merken om gesynchroniseerd te worden. Bij het uitvoeren van het multiprogramma verloopt het uitvoeren van dit aangemerkte deel dan via de monitor -- een thread of proces dat de aangemerkte code uit wil voeren, meldt zich aan bij de monitor en mag pas verdergaan als de monitor toestemming geeft. De monitor zorgt ervoor dat ten hoogste een thread of proces op ieder moment toestemming heeft. Door de algemene manier van werken van een monitor, is de controle over de synchronisatie meestal groffer dan bereikt kan worden via direct gebruik van seinpalen. Toch is de monitor zeer populair, aangezien synchronisatie van componenten in een multiprogramma door veel programmeurs als moeilijk wordt ervaren.

Hoofdstuk 3: Concurrency – parallele processen

1. Wat is concurrency?
2. Wederzijdse uitsluiting (mutual exclusion)
3. Het programmeren van wederzijdse uitsluiting
4. Het algoritme van Dekker
5. Het algoritme van Peterson
6. Semaforen
7. Monitoren
8. Synchronisatie
9. Deadlock
10. Threads

8. Synchronisatie

Het gemak waarmee semaforen wederzijdse uitsluiting afdwingen, maakt het ons mogelijk ook andere problemen, bijvoorbeeld het **synchroniseren van processen**, op te lossen.

We definiëren dit als het opleggen van een dwingende volgorde aan events die door concurrente, asynchrone processen worden uitgevoerd.

Wij moeten garanderen dat processen in een bepaalde volgorde verlopen.



Ter info wiki: Synchronisatie is het proces of het resultaat van iets gelijktijdig maken. Het is afgeleid van het Griekse σύν (sýn) 'samen' en χρόνος (chrónos) 'tijd'. Pas in de negentiende eeuw is synchronisatie in de geschiedenis van de mens een rol van betekenis gaan spelen. De treinen gingen zo snel rijden dat een verschil in lokale tijd op ging vallen. Het gelijk zetten van de klokken langs de spoorlijn werd noodzakelijk. Synchronisatie van de klokken was ook een vereiste voor het veilig gebruik van enkelspoor. De treinen konden zo volgens het spoorboekje blijven rijden, zodat vermeden werd dat twee treinen tegelijkertijd op hetzelfde spoorvak op elkaar aanstormden. Sindsdien is het belang van synchronisatie alleen maar groter geworden. In de informatica wordt de problematiek van synchroniseren geïllustreerd door het zogenoemde filosofenprobleem.

8. Synchronisatie (2)



Illustratie van de problematiek van synchroniseren a.d.h.v. het filosofen probleem:

De vijf filosofen zitten aan een tafel en kunnen twee dingen doen: spaghetti eten of filosoferen. Als ze eten kunnen ze niet denken en als ze denken kunnen ze niet eten. De spaghetti staat midden op de ronde tafel en om te eten heeft elke filosoof **twee** vorken nodig. Er zijn echter **slechts vijf** vorken. Zo heeft elke filosoof één vork aan zijn linker en één aan zijn rechterhand; de filosoof kan die oppakken, maar alleen een voor een.

Het probleem is nu om de filosofen zodanige instructies te geven dat ze niet zullen verhongeren.

Dit soort problemen zijn in het algemeen niet zo eenvoudig op te lossen.

**HO
GENT**

8. Synchronisatie (3)

Stel bijvoorbeeld dat elke denker als filosofie heeft: ik pak een vork zo gauw ik kan, als beide beschikbaar zijn eerst de linkervork; zo gauw ik beide vorken heb eet ik wat; dan leg ik de vorken weer neer.

Op het eerste gezicht een redelijk plan, maar nu kan de situatie ontstaan dat elke filosoof de linkervork in de linkerhand heeft, eeuwig wachtend tot de rechtervork vrijkomt. Dit is een voorbeeld van '**deadlock**': er is helemaal geen voortgang in het systeem meer mogelijk. Elke filosoof zal verhongeren.

Er zijn technieken om tot oplossingen te komen die deadlock bewijgbaar voorkomen; Dijkstra heeft het probleem verzonden om zulke technieken te demonstreren.

We kunnen bijvoorbeeld de denkers nummeren en elke denker alleen een vork laten pakken als er geen hoger genummerde denker al een vork vastheeft. Nu is deadlock onmogelijk.

**HO
GENT**

8. Synchronisatie (4)

Deadlock is echter niet het enige soort situatie dat in het ontwerp moet worden uitgesloten.

Stel bijvoorbeeld dat we een denker zelfs geen vork laten pakken als tegelijk een hoger genummerde hetzelfde probeert. Dan zal de hoogstgenummerde altijd eten, terwijl de rest verhongert. Zo'n situatie wordt **starvation** genoemd.

We kunnen dit nog verder aanscherpen, bijvoorbeeld door te eisen dat het systeem **eerlijk** is, in de zin dat de filosofen niet alleen allemaal altijd nog ooit de kans krijgen te eten, maar ze die kans zelfs even vaak krijgen; of door te eisen dat de totale wachttijd zo klein mogelijk is.

**HO
GENT**

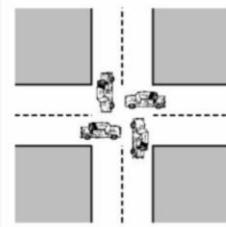
8. Synchronisatie (5)

Deze situatie illustreert de problemen die zich kunnen voordoen bij het synchroniseren van toegang tot resources (de vorken), bijvoorbeeld door verschillende threads (de filosofen) in een computerprogramma.

Als verschillende threads gebruik maken van dezelfde variabelen of bestanden is het niet veilig dat ze die tegelijk proberen aan te passen; daarom kan het onvermijdelijk zijn dat threads op elkaar moeten wachten. Als deze synchronisatie niet correct wordt ontworpen kan het voorkomen dat een thread helemaal nooit meer aan de beurt komt (**starvation**) of dat dat zelfs voor elke thread geldt (**deadlock**).

HO
GENT

9. Deadlocks



Een deadlock of een impassestoestand treedt op wanneer 2 of meer processen voor onbepaalde tijd wachten op een gebeurtenis die alleen door 1 van de wachtende processen kan worden veroorzaakt.

In principe zijn **er 2 methoden voor het behandelen van deadlock:**

- gebruik één of ander protocol (afspraak) om te garanderen dat het systeem nooit in een deadlock-situatie zal komen;
- laat toe dat het systeem in een deadlock-situatie geraakt en los deze dan op

**HO
GENT**

9. Deadlocks (2)

Voornaamste aspecten van deadlock:

→ Deadlock-preventie

Het besturingssysteem beperkt het gemeenschappelijk gebruik van resources om deadlock onmogelijk te maken.

→ Deadlock-vermijding

Het besturingssysteem onderzoekt alle aanvragen voor resources heel nauwkeurig. Ziet het besturingssysteem dat de toewijzing van een resource het risico van deadlock met zich meebrengt, dan weigert het de gevraagde toegang en vermijdt zo het probleem.

**HO
GENT**

9. Deadlocks (3)

Voornaamste aspecten van deadlock:

→ Deadlock-signalering

Als er een deadlock optreedt, moet het besturingssysteem dit kunnen signaleren. Het besturingssysteem ziet elk proces in een wachttoestand. Hoe kan het besturingssysteem erachter komen dat dit wachten permanent is?

→ Deadlock-herstel

Wat moet er gebeuren nadat het besturingssysteem een deadlock ontdekt? De processen moeten daar toch een keer uit bevrijd worden. Het besturingssysteem moet dit probleem oplossen.

**HO
GENT**

9. Deadlocks (4)

9.1. Deadlock-preventie

Een deadlock kan alleen dan optreden indien er tegelijkertijd aan de volgende 4 voorwaarden is voldaan:

- wederzijdse uitsluiting (mutual exclusion)
- bezet houden en wachten (hold en request)
- geen voortijdig ontnemen (non-preemption)
- wachten in een kring (circular wait)

Om deadlock te voorkomen zorgen we ervoor dat tenminste 1 van de voorwaarden nooit optreedt.

Er ontstaan aanzienlijke problemen als we proberen deadlock te voorkomen door een noodzakelijke conditie op te heffen.



Wederzijdse uitsluiting: Gemeenschappelijk gebruik van de resources moet onder wederzijdse uitsluiting plaatsvinden, d.w.z. als een proces tot een resource toegang heeft, mag geen enkel ander proces deze benaderen tot de resource is vrijgegeven.

Bezet houden en wachten. Een proces kan meerdere resources aanvragen zonder de eerder toegewezen resources vrij te geven. Er moet dus een proces bestaan dat ten minste 1 resource bezet houdt en dat tevens wacht op het verkrijgen van nog andere resources die op dat ogenblik door andere processen bezet zijn.

Geen voortijdig ontnemen. Resources kunnen niet voortijdig worden afgenomen d.w.z. dat een resource alleen vrijwillig kan worden vrijgegeven door het proces dat deze resource in bezit heeft, nadat het proces zijn taak heeft beëindigd.

Wachten in een kring. Er moet een verzameling $\{p_0, p_1, \dots, p_n\}$ van wachtende processen bestaan zodanig dat p_0 wacht op een resource dat in het bezit is van p_1 , p_1 wacht op een resource dat in het bezit is van p_2 , enz..., p_n wacht op een resource dat in het bezit is van p_0 .

Problemen:

→ Als er geen wederzijdse uitsluiting wordt afgedwongen, kunnen de activiteiten van het ene proces de voortgang van het andere proces beïnvloeden. Deze conditie mag dus niet worden verwijderd.

9. Deadlocks (5)

Problemen:

- Als er geen wederzijdse uitsluiting wordt afgedwongen, kunnen de activiteiten van het ene proces de voortgang van het andere proces beïnvloeden. Deze conditie mag dus niet worden verwijderd.
- Alvorens met zijn uitvoering te beginnen moet elk proces al de resources, die het nodig heeft, verkrijgen. Als een proces alle resources tegelijkertijd moet aanvragen, heeft het een aantal resources voor lange tijd onder beheer zonder ze daadwerkelijk te gebruiken. Dit beperkt de beschikbaarheid van de resources.
- Als het proces enkele resources vasthoudt en het vraagt nog een resource aan, en deze resource kan niet onmiddellijk aan dat proces worden toegewezen (d.w.z. het proces moet wachten), dan moet het proces al de resources die het op dat ogenblik vasthoudt, vrijgeven. Als we deze conditie verwijderen, dan kan een resource met geweld van een proces ontnomen worden.
- Onderwerp alle processen aan een lineair ordeningsschema. Ieder proces kan alleen resources in opklimmende volgorde verkrijgen.

**HO
GENT**

9. Deadlocks (6)

9.2. Deadlock-vermijden

Het verschil tussen het vermijden en het voorkomen van een deadlock is dat in het eerste geval deadlock niet onmogelijk is. Het idee is de aanvragen die eventueel tot deadlock kunnen leiden, te weigeren.

9.3. Deadlock signaleren

Steeds wanneer een proces een resource aanvraagt is er deadlock mogelijk. We vragen ons af hoe het besturingssysteem deadlock kan signaleren en wat het besturingssysteem eraan doet als er een deadlock is.

Eén manier om deadlock te signaleren, is een resource allocation graf. Dit is een georiënteerde graf die gebruikt wordt om de resource-toewijzingen weer te geven.

Een deadlock kunnen we signaleren door de resource allocation graf te bekijken. Als deze een cyclus bevat, is er een deadlock. Om cycli in een georiënteerde graf te signaleren, heeft het besturingssysteem diverse algoritmen ter beschikking.

**HO
GENT**

9. Deadlocks (7)

10.4. Herstel in een deadlock-situatie

Nu we weten hoe we een deadlock signaleren, rest ons nog 1 vraag: wat doen we eraan?

Eén mogelijkheid is een proces gewoon maar af te breken en de eraan toegewezen resources verwijderen. Hierdoor wordt de cyclus en dus ook de deadlock geëlimineerd ten koste van het proces.

Een andere mogelijkheid is een **rollback** op het proces uit te voeren. Hierbij worden alle eraan toegewezen resources verwijderd. Het proces verliest alle updates die het met gebruik van deze resources heeft gemaakt, en al het werk dat inmiddels was gedaan, maar wordt niet afgebroken. Het besturingssysteem brengt het terug in de toestand van vóór de aanvraag en toewijzing van de verwijderde resources. Dit kan overeenkomen met de oorspronkelijke start van het proces, of met een checkpoint. Een checkpoint treedt op wanneer een proces vrijwillig alle resources vrijgeeft. Door het gebruik van checkpoints kan elk proces eventueel verlies van werk echter zo klein mogelijk houden.

**HO
GENT**

Hoofdstuk 3: Concurrency – parallele processen

1. Wat is concurrency?
2. Wederzijdse uitsluiting (mutual exclusion)
3. Het programmeren van wederzijdse uitsluiting
4. Het algoritme van Dekker
5. Het algoritme van Peterson
6. Semaforen
7. Monitoren
8. Synchronisatie
9. Deadlock
10. Threads

10. Threads

Een proces bestaat uit 2 afzonderlijke en mogelijk onafhankelijke concepten: een concept dat samenhangt met de eigendom van bronnen en een concept dat samenhangt met de uitvoering. Dit onderscheid heeft in enkele besturingssystemen geleid tot de ontwikkeling van een constructie die bekendstaat als een **thread**.

Om een onderscheid te maken tussen de 2 concepten, wordt de eenheid voor de verdeling (uitvoering) doorgaans een thread of lichtgewicht proces genoemd en de eenheid voor de eigendom van bronnen een proces of een taak.

Multithreading verwijst naar de mogelijkheid van een besturingssysteem binnen een proces meerdere threads of draden te gebruiken voor de uitvoering. De traditionele benadering met één uitvoeringsthread per proces, waarin het concept thread in feite niet bestaat, wordt ook wel een benadering met één thread genoemd.



10. Threads (2)

In een omgeving met multithreading wordt een proces gedefinieerd als beveiligings- en brontoewijzingseenheid.

Het volgende is verbonden met processen:

- een **virtuele adresruimte**, die het procesbeeld bevat
- **beveiligde toegang** tot processors, andere processen (voor de communicatie tussen processen), bestanden en I/O bronnen (apparaten en kanalen)



10. Threads (3)

Binnen een proces kunnen er een of meer threads zijn, elk met het volgende (eigenschappen van threads):

- een **uitvoeringstoestand** van de thread (actief, gereed, ...)
- een **context** die wordt opgeslagen als de thread niet actief is, een thread kan onder meer worden gezien als een onafhankelijke programmateller die binnen een proces werkt
- **een stack** voor de uitvoering
- **enige statische opslagcapaciteit per thread** voor lokale variabelen
- toegang tot het geheugen en de bronnen van het bijhorende proces, die wordt gedeeld door alle threads binnen dat proces.

Alle threads van een proces delen de toestand en de bronnen van dat proces. Ze bevinden zich in dezelfde adresruimte en hebben toegang tot dezelfde gegevens.



10. Threads (4)

De grootste voordelen van threads hangen samen met de gevolgen voor de prestaties: het creëren van een nieuwe thread binnen een bestaand proces kost veel minder tijd dan het creëren van een geheel nieuw proces en ditzelfde geldt voor het overschakelen tussen 2 threads binnen hetzelfde proces.

Threads verbeteren ook de efficiëntie van de communicatie tussen verschillende actieve programma's. Aangezien threads binnen hetzelfde proces echter geheugen en bestanden delen, kunnen ze rechtstreeks met elkaar communiceren.

Net zoals processen hebben threads uitvoeringsstoestanden en kunnen ze met elkaar worden gesynchroniseerd.

**HO
GENT**